SUPPLEMENT TO

British Telecommunications Engineering

ISSN 0262-4028

Vol. 7 Part 1 April 1988

AN INTRODUCTION TO 16 bit MICROPROCESSORS

Part 1—8086/8088 Microprocessors

P. A. QUINSEY†

† Head of Micro-electronics, Matthew Boulton College, Birmingham

Contents

INTRODUCTION

Many engineers and technicians will already be familiar with one or more 8 bit microprocessors. These have become well established since the mid-1970s, but a number of applications demand higher performance.

The 16 bit microprocessor can overcome many of the limitations of an 8 bit design. However, most 16 bit microprocessors are very much more than 8 bit designs with double-length registers. Many 16 bit microprocessors have internal architectures very different from the traditional 8 bit design. Advances in integrated circuit fabrication techniques have now allowed much higher degrees of integration so that many more components can be formed upon a single chip.

The limitations of most 8 bit microprocessors include:

(*a*) Data words exceeding 8 bits may only be operated upon by splitting into 8 bit portions. This increases the execution time considerably.

(*b*) An increasing number of applications require more memory space than is directly available to an 8 bit microprocessor (typically 64 Kbyte).

(*c*) Most 8 bit microprocessors cannot perform multiplication and division directly. These tasks must then be undertaken by a subroutine.

(*d*) Most 8 bit microprocessors were not designed with multiple processor operation or resource sharing in mind. These devices do not have the necessary standard bus control signals readily available.

16 bit microprocessors have been designed with speed of execution as an important feature. Most use a pipelining technique to increase the speed of operation. This allows instructions to be fetched before execution of the previous instruction is completed. A further means of increasing the speed of operation is the use of cache memory. This is a special section of memory which provides fast local storage for the more frequently required data and code.

This paper, which will comprise two parts, introduces 16 bit microprocessors and the techniques involved by discussing two common families of 16 bit microprocessors: the Intel 8086/8088 family, which forms the basis of this first part, and the Motorola 68000, which will be described in Part 2.

8086 OVERVIEW

Intel introduced the 8086 (now also known as the *iAPX86/10*) in 1978. This device was one of the earliest available 16 bit microprocessors and is still in common use. Both the 8086 and the closely-related 8088 are capable of 16 bit internal transfers and can access 1 Mbyte of memory. The 8086 has a 16 bit external data bus, whereas the 8088 data bus is restricted to 8 bits externally. Clearly then, the 8088 is limited to 8 bit transfers with memory and input/output (I/O). This is the main difference between these two devices as far as the user is concerned.

Both the 8086 and the 8088 are designed to cover a range of applications, from simple minimal systems up to large multiple-processor systems. To support the more complex applications, several co-processors are available to enhance the capabilities of the system. These include the 8089 input/output processor and the 8087 maths co-processor.

One of the most notable applications in recent years has been the IBM personal computer (IBM PC). The basic machine and clones use the 8088 microprocessor. The higher performance and later models (for example, IBM PS/2 range) use different processors. These are however, downwardly compatible with the 8088/8086 machines.

Clock

The standard 8086 uses a 5 MHz clock, usually provided by the 8284 clock generator device. There are other clock speed versions available, from 4 MHz up to 10 MHz.

Input/Output

Up to 64K 8 bit input/output ports may be addressed. Both 8 and 16 bit words may be transferred and block transfer can also be used.

Instruction Set

The 8086/8088 processors have an instruction set of which the 8085 and 8080 instruction sets form sub-sets. This allows assembly language programs written for the 8085/8080 to be run successfully on the 8086.

Arithmetic instructions allow addition, subtraction, multiplication and division. Logical instructions allow AND, OR, EXCLUSIVE-OR and NOT functions to be used. Further instructions include INCREMENT/DECREMENT, COMPARE, COMPLEMENT, NEGATE, SHIFT/ROTATE and CLEAR.

Data transfer instructions include general-purpose data transfer instructions (for example, MOVE (LOAD)), input/output instructions and flag transfer instructions.

Program transfer instructions allow both conditional (for example, JUMP IF CARRY) and unconditional (for example, JUMP) transfers. Iteration control and software interrupt instructions are also provided.

Memory Segmentation

The 8086 address bus is 20 bits wide, but all 8086 internal registers are only 16 bits wide. 20 bit addresses are then formed by combining a number of internal registers.

The 20 bit physical address is formed by shifting a segment register left by four places and then adding the contents of a pointer register. Consider the example shown in Figure 1.

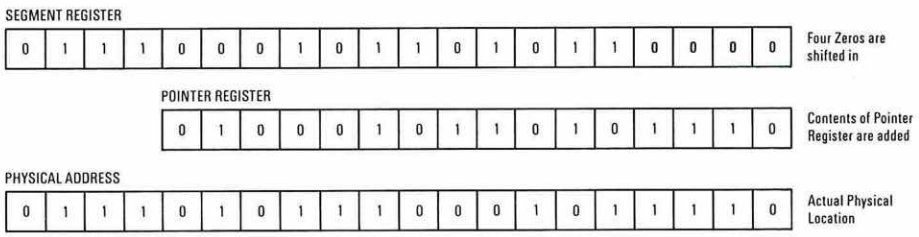


Figure 1
Example of 8086 addressing

The segment register originally contained:

$$0111000101101011_2 \quad (716B_H).$$

This register was then shifted left four places, and so four zeros shift in from the right, giving:

$$01110001011010110000_2 \quad (716B0_H).$$

The pointer register, containing

$$0100010110101110_2 \quad (45AE_H)$$

was then added to the shifted segment register to arrive at the final physical address:

$$01110101110001011110_2 \quad (75C5E_H).$$

8086 ARCHITECTURE

The basic 8086 internal architecture is shown in Figure 2.

There are two major sections within the CPU:

(*a*) the bus interface unit, and
(*b*) the execution unit.

These units work independently of each other, the bus interface unit being concerned with external operations (for example, instruction fetch), whilst the execution unit is involved only in the execution of instructions.

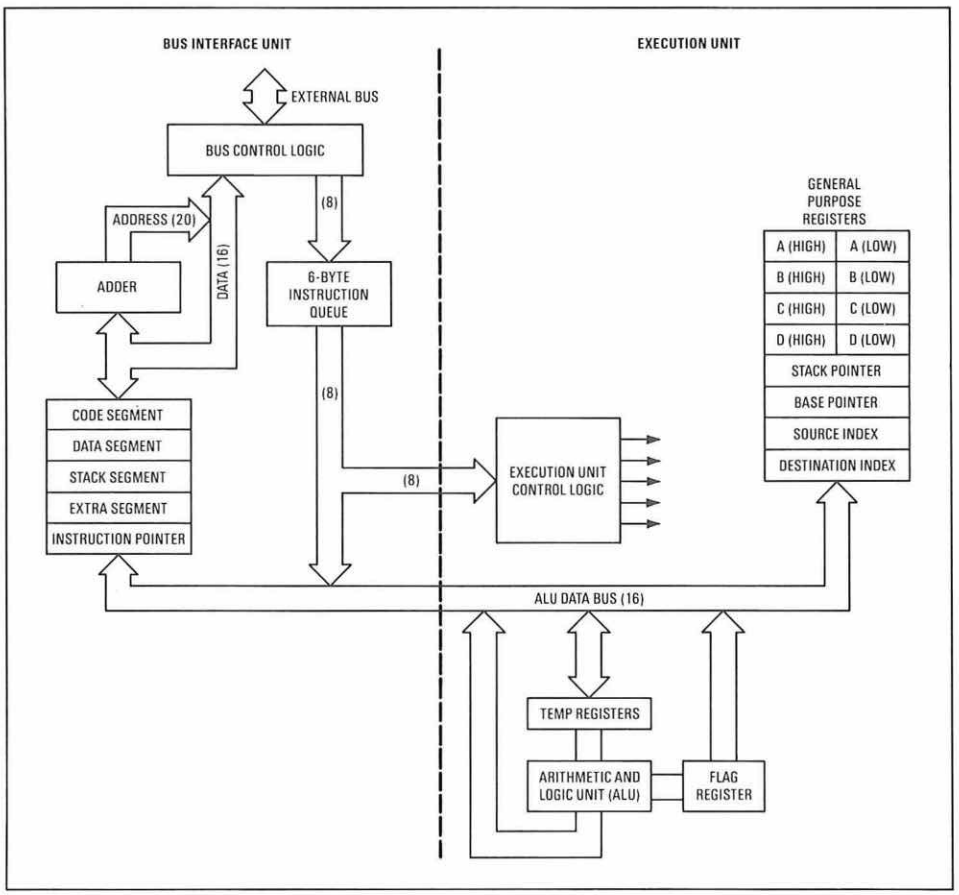


Figure 2
8086 internal architecture

Bus Interface Unit

The execution unit performs memory reads and writes via the bus interface unit. However, the bus interface unit also fetches instructions from memory whenever there is space in the instruction queue and the execution unit is not requesting bus access. If there are two or more bytes empty within the instruction queue, a 16 bit word is fetched from memory. The instruction queue is re-initialised by any program transfer instruction (for example, JUMP).

If the queue is full and the execution unit does not require access to the bus, then there are periods of time when the bus has no activity. These are called *bus idle states*. Such periods may be due to a number of causes, including the execution of a very long instruction and in multiprocessor systems.

This structure effectively eliminates CPU delays due to instruction fetches, although the execution unit is required to wait if branching out of the instruction queue sequence occurs (for example, JUMP).

SEGMENT REGISTERS

The segment registers are used to define 64 Kbyte blocks within memory.

Code Segment Register The contents of the program counter are added to the code segment register during an instruction fetch. This allows the physical location of the instruction to be calculated by the processor.

Data Segment Register The data segment register is used to define the physical location of data for all instructions except those involving stack or string operations.

Stack Segment Register Any instructions which involve stack operations use the stack segment register to define the physical location required.

Extra Segment Register The extra segment register is used in conjunction with the destination index register to define physical addresses for string operations.

Execution Unit

All execution tasks are performed by this unit. Although it always makes bus requests through the bus interface unit, the operation of the two units is independent.

The general-purpose registers are shown as two 8 bit registers. However, these can be addressed as 16 bit registers thus:

AX: A(High)—8 MSB, A(Low)—8 LSB
BX: B(High)—8 MSB, B(Low)—8 LSB
CX: C(High)—8 MSB, C(Low)—8 LSB
DX: D(High)—8 MSB, D(Low)—8 LSB

Note: MSB—most significant bits; LSB—least significant bits.

Thus, both 8 and 16 bit operands can be supported.

These registers, although nominally general purpose, are also assigned individual functions:

AX Register This register is the primary accumulator. All input/output operations are performed via this register.

BX Register The BX register is the base register. It is used in the calculation of memory addresses in certain addressing modes.

CX Register This is the count register. The CX register is decremented by string and loop operations, and so may be used to determine the number of passes through a program loop.

DX Register The DX register is the data register. It is used for indirect input/output port addressing.

The other registers in this section are essentially pointer registers, used to define the address of data.

The stack pointer register, in conjunction with the stack segment register, defines the address of the current top of the stack. The base pointer register defines addresses within the stack segment. The source and destination index registers are used to access data within data memory. These registers can be auto-incremented or auto-decremented on each pass through a loop and so are frequently used in string operations.

FLAG REGISTER

Figure 3
8086 status register

The 8086 has a 16 bit status register, shown in Figure 3.

Carry Flag (C) This flag is conditioned by arithmetic instructions. It is also affected by some SHIFT and ROTATE instructions. If the result of an addition exceeds 16 bits, the carry flag is set. In subtraction, the carry flag represents a borrow. It is always set following subtraction, unless a borrow is generated.

Parity Flag (P) The parity flag is conditioned by the least significant 8 bits of all data operations. If these bits contain an even number of 1's, the parity flag is set. An odd number of 1's clears this flag.

Auxiliary Carry Flag (A) This flag is set by a carry from bit 3 to bit 4 of the result of an arithmetic operation. It corresponds to the 8080/8085 auxiliary carry flag.

Zero Flag (Z) The zero flag is set whenever the result of a data operation is zero and is cleared otherwise.

Sign Flag (S) The sign flag follows the MSB of the result of an arithmetic operation. If the MSB is '0', the result is positive and the sign flag is cleared. A '1' in the MSB indicates that the result is negative, and so the sign flag is set.

Trap Flag (T) If this flag is set, the processor is placed in a single-step mode. This can be used for software debugging.

Interrupt Enable Flag (I) If the interrupt enable flag is set, then maskable interrupts are enabled. Such interrupts are ignored if this flag is clear.

Direction Flag (D) The source index and destination index registers may be auto-incremented or auto-decremented for string operations. The direction flag determines which of these takes place. If the direction flag is set, then the index registers auto-decrement. If this flag is clear, then auto-increments take place.

Overflow Flag (O) This flag is the EXCLUSIVE-OR of carrys into and out of the MSB in arithmetic operations. This represents a magnitude overflow in signed arithmetic.

8086 INSTRUCTION SET

The 8086 instruction set can be grouped for convenience into six categories:

- data transfer instruction,
- bit manipulation instructions,
- arithmetic instructions,
- processor control instructions,
- program transfer instructions, and
- string primitive instructions.

Data Transfer Instructions

Data transfer instructions are associated with the transfer of 8 or 16 bit data between registers and memory locations. A number of these instructions have two operands: a *destination* and a *source*. The source operand defines the location of data prior to transfer, while the destination operand specifies the location of data after transfer.

This group of instructions can be further divided into four sub-groups:

GENERAL PURPOSE

General purpose instructions are concerned with both general data transfers and with stack operations.

This sub-group includes the following instructions:

MOV Duplicates the data contained within one register into another register or memory location and vice versa.

For example:

MOV AX,DX	Copies the contents of the DX register into the AX register.
MOV DX,0021H	Places the value 0021_H in the DX register.

PUSH Copies the contents of a register or a memory location onto the current top of the stack.

For example:

PUSH AX	Saves the contents of the AX register in the current top stack location.

POP Copies the contents of the current top stack location into a register or a memory location.

For example:

POP DX	Restores the contents of the DX register from the current top stack location.

XCHG Swaps the contents of a register or memory location with another register.

For example:

XCHG AX,BX	Contents of AX and BX registers are interchanged.

INPUT/OUTPUT

Input/output instructions perform 8 or 16 bit data transfers between the 8086 system and external devices or systems.

The port addresses may be stated explicitly or be pointed to by the DX register.

This sub-group includes the following instructions:

IN Duplicates the data presented at a specified data input port within the accumulator.

For example:

IN AX,08H	Reads 16 bit data from port 08_H into the AX register.

OUT Duplicates the contents of the accumulator within a specified data output port.

For example:

OUT AX,DX	Outputs 16 bit data from the port specified by the contents of the DX register.

ADDRESS OBJECT

Address object instructions are associated with the calculation of physical addresses using segment registers.

The following instruction is typical of this sub-group:

LDS Duplicates the contents of a specified memory location within the data segment register.

For example:

LDS SI,[20H] — The operand 20_H is added to the data segment register, which then points to the location from which the source index register is to be loaded. The contents of the following location are placed within the data segment register.

FLAG TRANSFER

The flag transfer sub-group of instructions is concerned with the transfers of the status register.

This sub-group comprises the following instructions:

LAHF Copies the 8 LSBs of the flag register into the A(High) register.

SAHF Copies the contents of the A(High) register into the 8 LSBs of the flag register.

PUSHF Copies the contents of the flag register into the current top stack location.

POPF Copies the contents of the current top stack location into the flag register.

Bit Manipulation Instructions

The bit manipulation group of instructions comprises three sub-groups:

LOGICAL

Logical instructions allow the logical operators to be applied to 8 or 16 bit stored data.

The following instructions make up this sub-group:

AND Performs the logical AND of the contents of a register with the contents of a specified register or memory location. The result of this operation is stored within the destination register/location.

For example:

AND AX,BX — Performs the logical AND of the AX and BX registers, and places the result in the AX register.

OR Performs the logical OR of the contents of a register with the contents of a specified register or memory location. The result is stored in the location specified by the destination operand.

For example:

OR CL,37H — Performs the logical OR of the C (Low) register with the value 37_H. The result is stored in the C (Low) register.

XOR Performs the logical EXCLUSIVE-OR of the contents of a register with the contents of a specified register or memory location. The mechanism is similar to the AND and OR instructions.

NOT Inverts the contents of a specified register or memory location. No flags are affected by this instruction.

For example:

NOT BX — Suppose that the BX register had previously contained 5555_H. This instruction inverts each bit in turn; the contents of the BX register would be $AAAA_H$ after execution.

TEST Performs the logical AND of the contents of a register with the contents of a specified register or memory location. The result of this operation is lost and both source and destination are unchanged. However, the sign and zero flags are conditioned by this instruction.

For example:

TEST AL,55H	Performs the logical AND of the A(Low) register with the value 55_H. The sign and zero flags are conditioned but the contents of the A(Low) register are unchanged.

SHIFT

Both arithmetic and logical shifts of data can be performed by this set of instructions.

This sub-group includes the following instructions:

SAR Performs an arithmetic shift right (or multiple shifts right) of the contents of a specified register or memory location.

For example:

SAR AX	This shifts the contents of the AX register one place to the right.

SHL Performs a logical shift left (or multiple shifts left) of the contents of a specified register or memory location.

For example:

SHL AX,CL	This shifts the contents of the AX register left a number of times, as defined by the contents of the CL register.

ROTATE

The rotate instructions allow rotations of stored data to be performed. It is possible to rotate simply or through the carry flag.

This sub-group includes instructions such as:

ROL Performs a rotate left (or multiple rotates left) of the contents of a specified register or memory location.

RCR Performs a rotate right through carry (or multiple rotates right through carry) of the contents of a specified register or memory location.

Arithmetic Instructions

There are four distinct sub-groups within this group of instructions:

ADDITION

The addition instructions concern the execution of addition in pure binary, binary coded decimal (BCD) and ASCII formats.

This sub-group comprises the following instructions:

ADD Adds the contents of a register to the contents of a specified register or memory location.

For example:

ADD AL,[2000H]	This adds the contents of location 2000_H, within the current segment, to the contents of the A(Low) register; the result is placed within the A(Low) register.

ADC Adds the contents of a register to the contents of a specified register or memory location, plus the current state of the carry flag.

AAA Converts the contents of the A(Low) register into ASCII, after an ASCII addition.

DAA Converts the contents of the A(Low) register into BCD, after a BCD addition.

INC Adds 1 to the contents of a specified register or memory location.

SUBTRACTION

These instructions are associated with subtraction in pure binary, binary coded decimal (BCD) and ASCII formats.

The following instructions form this sub-group:

SUB Subtracts the contents of a specified register or memory location from the contents of a register.

For example:

SUB CX,1234H	This subtracts the value 1234_H from the CX register. The result is placed within the CX register.

SBB Subtracts the contents of a specified register or memory location from the contents of a register and then subtracts the complement of the current state of the carry flag from the result.

AAS Converts the contents of the A(Low) register into ASCII, after an ASCII subtraction.

DAS Converts the contents of the A(Low) register into BCD, after a BCD subtraction.

NEG Negates the contents of a specified register or memory location by subtracting the operand from zero, and using 2's complement arithmetic. It should be noted that this is not the same as the NOT instruction.

For example:

NEG AX	If the AX register had previously contained 5555_H, then the contents of the AX register would be $AAAB_H$ after the execution of this instruction.

DEC Subtracts 1 from the contents of a specified register or memory location.

CMP Subtracts the contents of a specified register or memory location from the contents of another register or memory location. The result is lost and the registers/memory locations are unaffected. This instruction is used to condition the flags.

MULTIPLICATION

The 8086 can perform 8 or 16 bit multiplication directly. Pure binary, signed binary and BCD multiplication are possible.

The instructions associated with multiplication are:

MUL Multiplies the contents of a specified register or memory location by the contents of the AX or A(Low) registers. Both multiplier and multiplicand are interpreted as pure binary numbers.

For example:

MUL AX,BX	This multiplies the contents of the AX and BX registers, and places the result within the AX register.

IMUL Multiplies the contents of a specified register or memory location by the contents of the AX or A(Low) registers. Both multiplier and multiplicand are interpreted as signed binary numbers.

AAM Adjusts the result of a BCD multiplication to give a BCD result.

DIVISION

Both 8 and 16 bit division of pure binary, signed binary and BCD data can be performed.

The instructions associated with division include:

DIV Divides the contents of the AX or A(Low) registers by the contents of specified register or memory location. Both numbers are interpreted as pure binary numbers.

For example:

DIV AX,BX	Divides the contents of the AX register by the contents of the BX register and places the result within the AX register.

IDIV Divides the contents of the AX or A(Low) registers by the contents of specified register or memory location. Both numbers are interpreted as signed binary numbers.

AAD Adjusts the result of a BCD division to give a BCD result.

Processor Control Instructions

There are essentially three types of processor control instruction:

FLAG OPERATIONS

Only the carry, interrupt enable and direction flags may be directly conditioned.

The carry flag can be set, cleared or complemented by using STC, CTC or CMC, respectively.

The interrupt enable flag can be set or cleared by STI or CLI, respectively.

The direction flag may also be set or cleared by using STD or CLD.

EXTERNAL SYNCHRONISATION

The external synchronisation instructions are associated with handshaking data transfers in single and multiple processor systems.

This sub-group comprises the following instructions:

HLT This stops program execution. Only an external interrupt or a reset restarts execution.

WAIT CPU activity is suspended by this instruction until a logic LOW is presented to the TEST pin.

ESC Places the contents of a specified memory location on the data bus. This is used in multi-processor applications.

LOCK This is used to prevent the 8086 from losing system bus control during the execution of an instruction. When a LOCK instruction is executed, the LOCK pin becomes a logic LOW and remains so during execution of the next instruction.

NO OPERATION (NOP)

The NOP instruction causes a time delay of three cycles and increments the program counter. This instruction can be used to give a very short time delay or as a dummy instruction.

Program Transfer Instructions

The program transfer instructions allow jumps and subroutine calls to be executed. Software interrupts and conditional software interrupts are also possible. This group can then be divided into four sub-groups:

CONDITIONAL

These instructions are used to allow decisions to be made during program execution, depending upon the result of the last arithmetic logic unit (ALU) operation.

All instructions in this sub-group have a common action, the only differences being in the conditions which are tested. Various flags within the status register can be tested and jumps taken if a flag or flags are set (that is, equal to 1) or clear (that is, equal to 0). If the specified condition is true, then program control is transferred to the address of the next instruction plus a signed displacement. This displacement is a two's complement value so both forward and backward conditional jumps are possible. If the condition is not true, then no action is taken and program execution continues from the next instruction in sequence.

This sub-group has a number of instructions, including:

JA/JNBE Jump if not below or equal. A jump by the specified displacement takes place only if the carry and zero flags are clear.

For example:

```
      MOV CL,04H
HERE  DEC CL
      JA HERE
```

In this short program section, the CL register is decremented until it reaches zero, whereupon the next instruction in the sequence is executed. This is a simple delay routine.

JE/JZ Jump if equal/jump on zero. A jump takes place if the result of the last ALU operation was zero (that is, zero flag set).

JNO Jump if no overflow. If the overflow flag is clear then a jump is executed.

JB/JNAE Jump if below/jump if not above or equal. This instruction tests the carry flag. The jump is taken only if the carry flag is set.

JCXZ Jump if CX zero. A jump takes place only if the contents of the CX register are equal to zero.

JLE/JNG Jump if less or equal/jump on not greater. A jump occurs if the zero flag is set or if the sign flag does not equal the overflow flag. Thus the jump is not taken if the zero flag is clear and the sign and overflow flags are the same.

It should be noted that a significant number of 8086 instructions do not affect any flags. It is essential to check the conditioning of flags with manufacturer's literature or an assembly language programming textbook.

UNCONDITIONAL

The instructions in the unconditional sub-group always transfer program execution out of sequence.

There are three instructions within this sub-group:

JMP This is the unconditional jump instruction. The destination for a jump can be specified by:

(*a*) an absolute address,
(*b*) an 8 bit displacement,
(*c*) a 16 bit displacement, or
(*d*) a register.

CALL This instruction is used to call a subroutine. The address of the subroutine can be specified by:

(*a*) an absolute address,
(*b*) a 16 bit displacement, or
(*c*) a register.

RET This is the RETURN instruction, used to restore the program counter at the end of a subroutine. There are a number of types of RETURN but all 'pop' the top two stack bytes into the program counter. The differences lie within the actions following this pop. The four RETURN mechanisms are:

(*a*) pop into program counter,
(*b*) pop into program counter and code segment register,
(*c*) pop into program counter and add displacement to the stack pointer, and
(*d*) pop into program counter and code segment register, then add displacement to the stack pointer.

ITERATION CONTROL

The instructions in the iteration-control sub-group all decrement the CX register and then take a conditional jump, by using a relative displacement to specify the destination.

LOOP Decrements the CX register and jumps if the contents of the CX register are non-zero.

LOOPE/ LOOPZ Decrements the CX register and jumps if the contents of the CX register are non-zero and the zero flag is set.

LOOPNE/ LOOPNZ Decrements the CX register and jumps if the contents of the CX register are non-zero and the zero flag is clear.

INTERRUPTS

There are three instructions associated with interrupts:

INT This instruction performs a software interrupt so that an interrupt which is synchronous with program execution occurs.

For example:

INT 20H This causes a type 20_H interrupt to occur.

INTO This is a conditional software interrupt. A software interrupt only occurs if the overflow flag is set, otherwise no operation is performed.

IRET This instruction allows the program counter to be restored from the stack after a software or hardware interrupt.

String Primitive Instructions

A string primitive executes an operation sequence usually performed by a program loop. A primitive operation is performed and then the appropriate pointer register is incremented/decremented, according to the state of the direction flag.

The pointer register is incremented if the direction flag is clear and decremented if the direction flag is set. Incrementing and decrementing by both 1 and 2 is possible, depending

upon the state of the LSB of the string primitive opcode. If this LSB is a 0, then single increments/decrements take place; a 1 results in double increments/decrements.

There are five string primitives available:

MOVS Moves 8 or 16 bit data from the location specified by the source index register within the data segment. The destination for this move is specified by the destination index register within the extra segment.

LODS Loads 8 or 16 bit data from the location specified by the source index register within the data segment. The destination for this move is the A(Low) or AX registers, respectively.

STOS Stores the contents of the A(Low) or AX registers within the memory location specified by the destination index register within the extra segment.

SCAS Compares the contents of the A(Low) or AX registers with the data contained within the memory location specified by the destination index register within the extra segment.

CMPS Compares the contents of the memory location specified by the source index register within the data segment with the data contained within the memory location specified by the destination index register within the extra segment.

8086 ADDRESSING

A programmer's model of the 8086 is shown in Figure. 4.

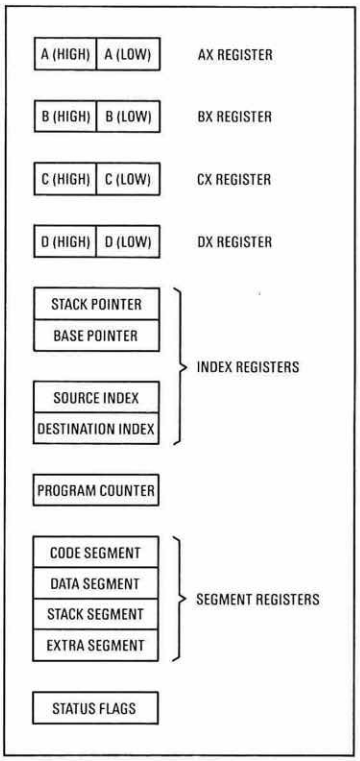


Figure 4
8086 registers

8086 Memory Addressing

Recall that any given physical address is formed from two addresses: the segment address (defined by the contents of the segment register) and the effective address (or offset address). The precise mechanism is not quite as simple as this suggests, since the segment register is shifted four places to the left prior to addition with the effective address.

Suppose that the contents of the segment register are:

$$PPPP\ PPPP\ PPPP\ PPPP_2 \text{ (or } PPPP_H)$$

and the effective address is:

$$QQQQ\ QQQQ\ QQQQ\ QQQQ_2 \text{ (or } QQQQ_H),$$

then the internal calculation is as follows:

Binary:						Hexadecimal:	
PPPP	PPPP	PPPP	PPPP	0000_2	+	$PPPP0_H$	+
0000	QQQQ	QQQQ	QQQQ	$QQQQ_2$		$0QQQQ_H$	
RRRR	SSSS	SSSS	SSSS	$TTTT_2$		$RSSST_H$	

The final physical address is then:

$$RRRR\ SSSS\ SSSS\ SSSS\ TTTT_2 \text{ (or } RSSST_H).$$

The effective address gives the address within a data segment, defined by the segment register. Each of the four segment registers defines the start of a 64 Kbyte segment.

For example:

If the segment register contains 5678_H and the effective address $9ABC_H$:

Binary:						Hexadecimal:	
0101	0110	0111	1000	0000_2	+	56780_H	+
0000	1001	1010	1011	1100_2		$09ABC_H$	
0110	0000	0010	0011	1100_2		$6023C_H$	

Here the final physical address is:

$$0110\ 0000\ 0010\ 0011\ 1100_2 \text{ (or } 6023C_H).$$

Now, the effective address may have any value between 0000_H and $FFFF_H$. It follows therefore that the physical address may have any value between the shifted segment register value and the shifted segment register value plus $FFFF_H$. So, for instance, if the segment register contains 1234_H, then the address range pointed to would be from:

$$(12340_H + 0000_H) = 12340_H$$
$$\text{to } (12340_H + FFFF_H) = 2233F_H.$$

The 8086 has four segment registers, and so four separate 64 Kbyte segments can be pointed to at any given instant. Also, as there is no restriction upon the values placed within these registers, it is possible for segments to overlap.

The physical address of an opcode is formed from the code segment register and the program counter. Therefore, the code segment of memory contains programs.

8086 Addressing Modes

The 8086 has a number of basic addressing modes. These may be combined in certain circumstances to give apparently complex but potentially very powerful addressing modes.

IMMEDIATE ADDRESSING

In immediate addressing, the data to be acted upon is contained within the code segment byte (or bytes) immediately following the opcode in memory (low byte first for 16 bit operands). Immediate addressing can be used whenever the data required is a constant.

For example:

MOV AX,1234H	This loads the hexadecimal value 1234_H into the AX register.
ADD BX,5678H	This adds the hexadecimal value 5678_H to the current contents of the BX register.
MOV DL,9AH	This loads the hexadecimal value $9A_H$ into the D (Low) register (that is, the least significant 8 bits of the DX register).

DIRECT ADDRESSING

In this addressing mode, the two bytes following the operator (low byte first) are added to the shifted data segment register to determine the address of the data to be acted upon. This addressing mode is used where the data is not a constant value.

For example:

MOV [1234H],AX	This stores the contents of the AX register in location 1234_H within the current data segment. Therefore, if the data segment (DS) = 5678_H, then $56780_H + 1234_H = 579B4_H$. Thus the contents of the AX register are saved in location $579B4_H$.
ADD BX,[8765H]	This adds the contents of memory location 8765_H within the current data segment to the contents of the BX register. Therefore, if DS = 4321_H, then $43210_H + 8765_H = 4B975_H$. Thus the contents of memory location $4B975_H$ are added to the BX register.
MOV AX,[4321H]	This loads the AX register from memory location 4321_H in the current data segment. Therefore, if DS = 5678_H, then $56780_H + 4321_H = 5AAA1_H$. Thus the AX register is loaded from physical location $5AAA1_H$.

DIRECT INDEXED ADDRESSING

The 8086 has two index registers: the source index register and the destination index register. Either of these registers can be used as an indexing pointer. The index register used can be incremented or decremented to point to the next location in memory. The index registers can also be loaded with an immediate value so that the area pointed at can be easily changed.

For example:

MOV AX,[SI]	This loads the AX register from the memory location pointed at by the source index (SI) register. So, if SI = 1234_H and the code segment CS = 5678_H, then $56780_H + 1234_H = 579B4_H$. Thus the AX register is loaded from the physical location $579B4_H$.
MOV [DI],BX	This stores the contents of the BX register in the location pointed at by the destination index (DI) register. So, if DI = 3456_H and CS = $789A_H$, then $789A0_H + 3456_H = 7BDF6_H$. Thus the BX register is stored in the physical location $7BDF6_H$.

It is also possible for the contents of another register to be added to the contents of the index register in order to arrive at the final location of data.

For example:

MOV AX,[BP+SI]	This loads the AX register from the memory location formed by adding the base pointer (BP) register to the source index register. So, if BP = 1234_H and SI = 5678_H, then $1234_H + 5678_H = 68AC_H$. Thus the AX register is loaded from location $68AC_H$ within the current segment.

The direct indexed mode of addressing also allows an 8 bit or 16 bit displacement to be added to the contents of the index register to specify the effective address.

For example:

MOV AX,10H[BP+SI]	The displacement 10_H is added to the sum of the base pointer and source index registers to arrive at the final address, within the current segment.

BASE RELATIVE ADDRESSING

This mode of addressing uses a displacement to specify the location of data within the current segment.

The base for the effective address is the contents of the BX register. Base relative addressing can also be applied to other modes (for example, direct indexed as already seen).

Base relative addressing adds the contents of the BX register to the effective address to generate the final address.

For example:

MOV AX,80H[BX]	This loads the AX register from the location formed by BX + 80_H.

The contents of the BX register may also be added to an index register to modify the final address.

For example:

MOV [BX+DI],AX	This stores the contents of the AX register in the location formed by BX + DI.

It is also possible for this mode of addressing to add the contents of the BX register to a direct indexed operand. This gives base relative direct indexed addressing.

For example:

MOV AX,80H[BX+SI]	This loads the AX register from memory location formed by 80_H + BX + SI.

STACK MEMORY

This addressing mode is similar to the base relative mode. However, in this mode, the base register is the base pointer register.

The stack memory address is formed by adding the base pointer register and the specified displacement. The shifted contents of the stack segment register are then added to produce the actual stack address required.

8086 ASSEMBLY LANGUAGE PROGRAMMING

A number of elementary assembly language programs are given here as an introduction to 8086 assembly language. For the sake of simplicity, the segment addresses have been ignored. This is a reasonable assumption, since very few programs that exceed 64 Kbyte will be written in assembly language. It is hoped that these very simple examples will provide a base upon which to build further expertise in programming the 8086. Further programming problems may be found in several 8086 assembly language programming textbooks, including:

(*a*) *The 8086 Book* by Rector and Alexy, and
(*b*) *Assembly Language Programming for the 8088/8086* by Leventhal *et al.*

Although the 8086 is a 16 bit processor, the memory is still byte (rather than word) orientated. Consecutive 8 bit memory locations are used as 16 bit locations. For example:

```
MOV AX,[2000H]
```

This instruction loads the AX register from 2000_H (low byte or A(Low) register) and 2001_H (high byte or A(High) register).

It is most important to note that there really is no substitute for practical experience in assembly language programming.

Typically, three distinct pieces of software are required:

(*a*) ***Text Editor*** This is used to write the assembly language program mnemonics (many word processors will be suitable).

(*b*) ***Assembler*** This converts the mnemonics to machine code and resolves references (for example, labels).

(*c*) ***Debugger*** This allows the user's program to be run single-stepped, etc. It also allows the contents of memory locations to be displayed and modified.

Many engineers and technicians now have access to an IBM-PC or clone. The original IBM-PCs were 8088-based and the profusion of these machines has led to all the above listed software modules being readily available. Few machines are without some form of text editor; many word processor packages are suitable, although it may be necessary to modify the mode of operation (for example, WordStar should be in the 'non-document' mode). The debug function is usually provided by DOS ('DEBUG'), so all that is probably required is an assembler. There are a number of 8086 assemblers available, varying in cost and facilities. Some impressive public domain assemblers are available at a very low cost.

Example 1:

A program which adds the values 1234_H and 5678_H.

Note: This type of problem is easily solved by using a straightforward approach, familiar to users of 8 bit microprocessors.

```
START    MOV AX,1234H      ;Loads the AX register with 1234H.
         ADD AX,5678H      ;Adds 5678H to the AX register.
         MOV [5000H],AX    ;Stores the result in location 5000H
                            within the current segment.
FINISH   JMP FINISH        ;Wait forever.
```

If this program is assembled then the following is produced:

```
0100 B8 34 12     START    MOV AX,1234H     ;Loads the AX register
                                             with 1234H.
0103 05 78 56              ADD AX,5678H     ;Adds 5678H to the AX
                                             register.
0106 A3 00 50              MOV [5000H],AX   ;Stores the result in
                                             location 5000H within
                                             the current segment.
0109 E9 FD FF     FINISH   JMP FINISH       ;Wait forever.
```

Notes:

(*a*) Direct addresses are presented low byte first (see instructions at 0100_H, 0103_H and 0106_H).

(*b*) The result of the addition is stored in locations 5000_H (low byte) and 5001_H (high byte) within the current memory segment.

(*c*) The instruction at 0190_H is an unconditional relative jump to itself. The displacement is a two's complement value, added to the program counter. Thus $FFFD_H$ represents a backward jump of 3 (since $FFFD_H$ represents -3_H).

Example 2:

A program which reads the value presented at port 80_H and outputs the inverse at port 82_H, assuming ports to be already initialised.

Note: The NOT instruction can be used to invert.

```
0100 E5 80          START     IN AX,80H        ;Read input port.
0102 F7 D0                    NOT AX           ;Invert.
0104 E7 82                    OUT 82H,AX       ;Output inverted value.
0106 E9 F7 FF                 JMP START        ;Loop forever.
```

Example 3:

A program which exchanges the contents of two 16 bit memory locations.

Note: The XCHG (exchange) instruction is the obvious one to use here. This function is not available on most 8 bit microprocessors.

```
0100 A1 00 50       START     MOV AX,[5000H]   ;Loads AX register from
                                                first location.
0103 87 06 02 50              XCHG AX,[5002H]  ;Exchanges the contents
                                                of the second location
                                                with those of the AX
                                                register.
0107 A3 00 50                 MOV [5000H],AX   ;Replaces the contents
                                                of the first location
                                                with those of the
                                                second.
010A E9 FD FF       FINISH    JMP FINISH       ;Wait forever.
```

Notes:

(*a*) The XCHG instruction allows register–register or register–memory exchanges, but not memory–memory exchanges.
(*b*) Notice that the XCHG instruction requires a 2 byte operator since the microprogram is quite complex.

Example 4:

A program which will multiply the contents of two memory locations.

Note: Here, the MUL (multiply) instruction can be used, rather than a 'shift and add' technique.

```
0100 A1 00 50       START     MOV AX,[5000H]   ;Loads the AX register
                                                from the first
                                                location.
0103 F7 26 02 50              MUL AX,[5002H]   ;Multiply AX register by
                                                the contents of the
                                                second location.
0107 A3 04 50                 MOV [5004H],AX   ;Store the result in 16
                                                bit location 5004H.
010A E9 FD FF       FINISH    JMP FINISH       ;Wait forever.
```

Notes:

(*a*) This program is very much shorter than a 'shift and add' algorithm implementation.
(*b*) Since it has a complex sequence of micro-instructions, the MUL instruction requires a 2 byte operator.

Example 5:

A program which divides the contents of one 16 bit location by another 16 bit location.

Note: Here, the DIV (divide) instruction can be used, rather than a 'shift and subtract' technique.

```
0100 A1 00 50       START     MOV AX,[5000H]   ;Loads the AX register
                                                from the first
                                                location.
0103 F7 36 02 50              DIV AX,[5002H]   ;Divides the AX register
                                                by the contents of the
                                                second location.
0107 A3 04 50                 MOV [5004H],AX   ;Store the result in 16
                                                bit location 5004H.
010A E9 FD FF       FINISH    JMP FINISH       ;Wait forever.
```

Notes:

(*a*) This program is very much shorter than a 'shift and subtract' algorithm implementation.
(*b*) The DIV instruction requires a 2 byte operator, since it has a complex sequence of micro-instructions.

Example 6:

A program which adds all the hexadecimal integers from 01_H to $2F_H$.

Note: This program clearly requires a loop. Use can be made of the auto-increment/auto-decrement facilities of the 8086.

```
0100 B8 00 00      START     MOV AX,0000H     ;Clears the AX register.
0103 B9 2F 00                MOV CX,002FH     ;Loads the CX register
                                               with the number of
                                               integers to be added.
0106 01 C8         CALCUL    ADD AX,CX        ;Adds Nth integer to
                                               accumulator.
0108 E2 FC                   LOOP CALCUL      ;Decrements the CX
                                               register and jumps to
                                               next addition if the CX
                                               register is non-zero.
010A A3 00 50                MOV [5000H],AX   ;Stores the result in 16
                                               bit location 5000H.
010D E9 FD FF      FINISH    JMP FINISH       ;Wait forever.
```

Note: The LOOP instruction combines a conditional jump with a register decrement.

Example 7:

A program which examines the contents of locations $5000/5001_H$ and $5002/5003_H$ and copies the smaller into location $5004/5005_H$.

Note: Here a CMP instruction can be used.

```
0100 A1 00 50      START     MOV AX,[5000H]   ;Read contents of first
                                               location.
0103 3B 06 02 50             CMP AX,[5002H]   ;Compare first and
                                               second locations.
0107 73 06                   JNC OTHER        ;Is first or second
                                               smaller?
0109 A3 04 50      STORE     MOV [5004H],AX   ;First is smaller, so
                                               store in 5004H.
010C E9 FD FF      FINISH    JMP FINISH       ;End.
010F A1 02 50      OTHER     MOV AX,[5002H]   ;Second is smaller, so
                                               load AX with second
                                               value.
0112 E9 F4 FF                JMP STORE        ;Save second value.
```

Notes:

(*a*) The CMP instruction subtracts the contents of location 5002_H from the AX register without changing the AX register contents.

(*b*) The JNC instruction tests the status of the carry flag following the CMP. This shows which value is the smaller.

Example 8:

A program which adds the contents of $5000/5001_H$ and $5002/5003_H$ and places the results in locations $5004/5005/5006_H$.

Note: A carry could be generated by this addition, so it is necessary to test for this possibility.

```
0100 A1 00 50      START     MOV AX,[5000H]   ;Load AX with first
                                               number.
0103 F8                      CLC              ;Clear carry flag.
0104 13 06 02 50             ADC AX,[5002H]   ;Add second number.
0108 A3 04 50                MOV [5004H],AX   ;Save result in 5004H.
010B 73 09                   JNC ZERO         ;Test for carry set.
010D B3 01                   MOV BL,01H       ;Carry set so load
                                               B(Low) with 01H.
010F 88 1E 06 50   STORE     MOV [5006H],BL   ;Store carry status in
                                               location 5006H.
0113 E9 FD FF      FINISH    JMP FINISH       ;End.
0116 B3 00         ZERO      MOV BL,00H       ;Carry is clear, so load
                                               B(Low) with 00H.
0118 E9 F4 FF                JMP STORE        ;Store carry status.
```

Notes:

The carry flag must be cleared prior to an ADC instruction.

Example 9:

A program which will add two 32 bit numbers. The first number is contained in locations $5000/5001/5002/5003_H$ and the second in locations $5004/5005/5006/5007_H$. The result is to be placed in locations $5008/5009/500A/500B/500C_H$.

Note: This is just an extention of the previous problem.

```
0100 A1 00 50      START     MOV AX,[5000H]  ;Load  AX with low  word
                                              of first value.
0103 FB                      CLC             ;Clear carry flag.
0104 13 06 04 50             ADC AX,[5004H]  ;Add low word of  second
                                              value.
0108 A3 08 50                MOV [5008H],AX  ;Save   low   word    in
                                              5008/5009H.
010B A1 02 50                MOV AX,[5002H]  ;Load AX with high  word
                                              of first value.
010E 13 06 06 50             ADC AX,[5006H]  ;Add high word of second
                                              value.
0112 A3 0A 50                MOV [500AH],AX  ;Save   high   word   in
                                              500A/500BH.
0115 73 09                   JNC ZERO        ;Test for carry status.
0117 B3 01                   MOV BL,01H      ;Carry   set   so   load
                                              B(Low) with 01H.
0119 88 1E 0C 50   STORE     MOV [500CH],BL  ;Save  carry  status  in
                                              500CH.
011D E9 FD FF      FINISH    JMP FINISH      ;End.
0120 B3 00         ZERO      MOV BL,00H      ;Carry  clear,  so  load
                                              B(Low) with 00H.
0122 E9 F4 FF                JMP STORE       ;Save carry status.
```

Example 10:

A program which fills locations 5000_H to $5FFF_H$ with the value 77_H.

```
0100 BF 00 50      START     MOV DI,5000H    ;Load   DI   with   start
                                              address of block.
0103 BB 77 00                MOV BX,77H      ;Load  BX with value  to
                                              be stored.
0106 89 1D         NEXT      MOV [DI],BX     ;Save in Nth location.
0108 47                      INC DI          ;Add  1  to  destination
                                              address.
0109 81 FF 00 60             CMP DI,5FFFH    ;Compare DI with 6000H.
010D 7E F7                   JLE NEXT        ;Test      for      loop
                                              completed.
010F E9 FD FF      FINISH    JMP FINISH      ;Completed, so end.
```

Note: This program can easily be modified to accomodate different values and fill areas.

8086 INTERRUPTS

8086 interrupts have a table of interrupt vectors, stored in memory locations 00000_H to $003FF_H$. Each vector has four bytes. The first two bytes define the new program counter (instruction pointer) value, while the second two define the new code segment register value. These are combined by the usual 8086 method, previously described, to produce a 20 bit physical address which is the start address of the interrupt service subroutine.

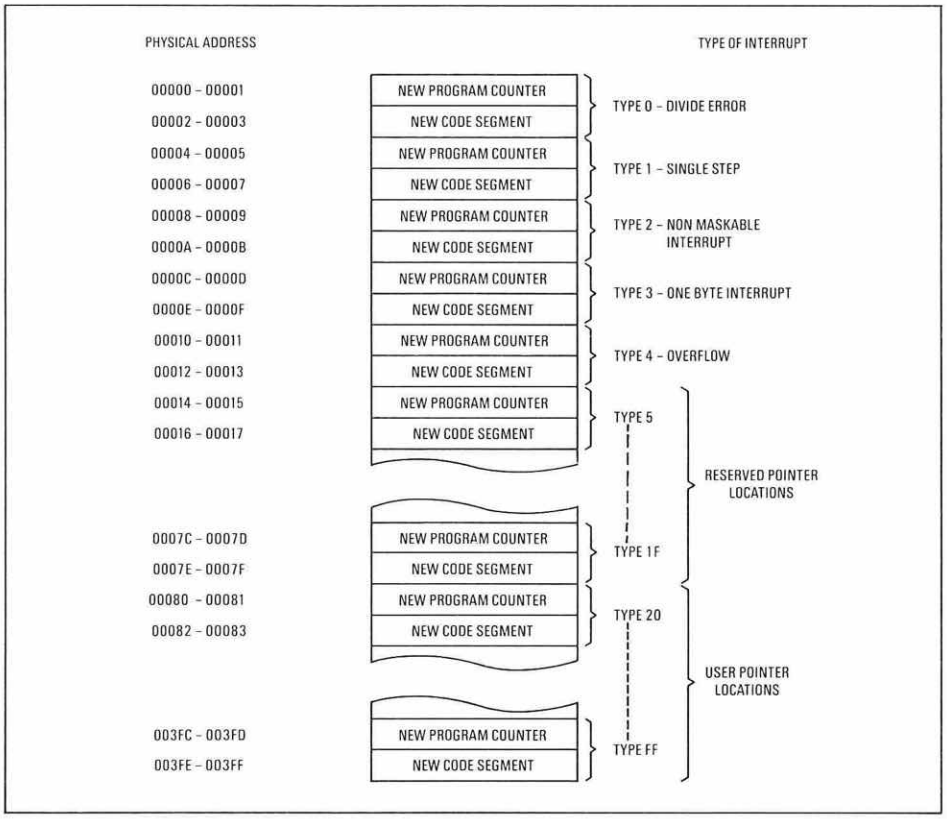
Each interrupt vector has a corresponding interrupt number, from 00_H to FF_H. This interrupt number, multiplied by 4 (shifted left by two places), gives the address of the first byte of that vector within the vector table. For example, interrupt 20_H:

$$20_H \times 04_H = 80_H.$$

The first byte of the vector is at location 00080_H. Thus the new program counter value is contained in locations 00080_H and 00081_H, while the new code segment value is contained in locations 00082_H and 00083_H.

A table of the interrupt vectors is shown in Figure 5.

Figure 5
8086 interrupt vectors



Interrupts may have been generated by an instruction (software interrupt) or by external logic (hardware interrupt).

The general interrupt response is listed below:

(*1*) The status register is pushed onto the stack.
(*2*) The interrupt enable flag is cleared, so that further interrupts are disabled.
(*3*) The code segment register and program counter are pushed onto the stack.
(*4*) The code segment register is loaded from the first two interrupt vector bytes.
(*5*) The program counter (instruction pointer) is loaded from the third and fourth interrupt vector bytes.
(*6*) Fetch and execute resumes at the start of the interrupt service subroutine.

At the end of the interrupt service subroutine, the main program may be resumed by the execution of a return from interrupt (IRET) instruction. This restores the status register, code segment register and program counter register from the stack.

The 8086 supports four types of interrupt:

(*a*) predefined interrupts,
(*b*) user-defined software interrupts,
(*c*) user-defined hardware interrupts, and
(*d*) restart.

Predefined Interrupts

Interrupt numbers 00_H to $1F_H$ are predefined interrupts. Types 0_H to 4_H have the special functions shown below:

Type 0 If the result of a division gives a result which is greater than the maximum allowable (typically divide by zero), then a type 0 interrupt is always requested. This interrupt is non-maskable.

Type 1 Whenever the trap flag is set, a type 1 interrupt occurs. This is used for single stepping of programs, which is a valuable aid in debugging.

Type 2 This is the only predefined hardware interrupt. A type 2 interrupt is a non-maskable interrupt. This interrupt input is triggered by a positive-going edge on the NMI input pin. Interrupt vectors for a type 2 interrupt cannot be directly supplied by an interrupting device.

Type 3 This is a 1 byte non-maskable software interrupt. Type 3 interrupts can be used to set breakpoints within user programs, as an aid in debugging.

Type 4 If the overflow flag is set when the INTO instruction is executed, then a type 4 interrupt occurs. This allows overflow error routines to be used.

Interrupt types 05_H to $1F_H$ are reserved as pointer locations, while interrupt types 20_H to FF_H are available as user pointers.

User Defined Software Interrupts

The INT instruction executes user software interrupts, types 20_H to FF_H.

User Defined Hardware Interrupts

This interrupt input is level triggered and maskable. Interrupting signals on this pin are only acknowledged if the interrupt enable flag has been set. Otherwise no interrupt response occur.

When a valid INTR signal occurs, the general interrupt response is followed. However, the interrupt number is provided by the interrupting device, rather than a predefined vector. The 8086 issues an *interrupt acknowledge* signal (INTA) which prompts the interrupting device to supply the interrupt number.

Restart

This is a system reset. When an active signal is presented at this pin, the following response occurs:

(*1*) The status register is cleared (that is, loaded with 0000_H). In particular, the interrupt enable flag and trap flag are cleared. This disables both interrupts and single-step mode.

(*2*) The following registers are also cleared:

(*a*) program counter,
(*b*) data segment,
(*c*) extra segment, and
(*d*) stack segment,

(*3*) The code segment register is loaded with $FFFF_H$.
(*4*) Fetch and execute begins from location $FFF0_H$.

8086 HARDWARE

The 8086 device is available as a standard 40-pin dual-in-line device. However, since 40 pins are inadequate for the number of signals required, some pin functions are multiplexed. This allows the total pin count to be reduced.

The pin configuration for the 8086 is shown in Figure 6.

Figure 6
8086 pin configuration

Address and Data Signals

AD0–AD15 — These pins carry data and the least significant 16 bits of address information. Address and data information is time-division multiplexed.

A16–A19 — The most significant 4 bits of address information is carried by these pins.

Processor Control Signals

CLK — This is the master timing signal for the synchronisation of all CPU operations.

RESET — A logic transition from HIGH to LOW on this pin causes a restart. The flags, DS, ES, and SS registers are cleared and the CS register set to $FFFF_H$. Fetch and execute resumes from location $FFFF0_H$.

READY — If a logic LOW is presented at this input, the CPU enters a WAIT state, where only internal 'wait' cycles are executed.

HOLD — A logic HIGH on this pin causes CPU activity to be suspended and outputs to be placed in a high impedance state. This may only occur after completion of the current bus cycle.

$\overline{TEST}$ — When a WAIT instruction is executed, the processor pauses until a logic LOW appears at this pin.

MN/$\overline{MX}$ — A logic HIGH on this pin selects the MINIMUM mode, whereas a logic LOW selects the MAXIMUM mode. The function of a number of 8086 pins changes according to the mode selected.

Data Transfer Control Signals

$\overline{RD}$ — This output becomes a logic LOW during a read of a memory or input/output (I/O) device.

$\overline{WR}$ — This output becomes a logic LOW during a write of a memory or I/O device.

ALE — A logic HIGH on this pin indicates that the current address/data bus information is a valid address.

DEN — A logic LOW on this pin indicates that the current address/data bus information is valid data. This can be used to enable bidirectional data bus buffers.

DT/$\overline{R}$ — This output can be used to control the direction of data transfers via bus transceivers. A logic HIGH indicates data flow away from the processor, while a logic LOW indicates flow towards the processor.

M/$\overline{IO}$ — This output pin indicates whether a memory or an I/O device is to be accessed. A logic HIGH indicates a memory read/write, and a logic LOW an I/O read/write.

Interrupt Control Signals

INTR — This is the maskable interrupt input. When a logic HIGH is presented at this pin and the interrupt enable flag is set, then program execution is transferred to an interrupt service routine, after completion of the current instruction.

INTA — This pin becomes logic LOW during an interrupt acknowledge sequence.

NMI — This is the non-maskable interrupt input and so may not be ignored. A transition from a logic LOW to HIGH on this pin causes a type 2 interrupt response.

Processor Status Signals

HLDA — This is a hold acknowledge output. This pin becomes a logic HIGH to acknowledge a *hold* input.

S3–S6 — These pins are time-division multiplexed. They convey address information during the first part of a bus cycle. During other periods, status information is produced. This indicates which segment register is being used to specify the segment address at any given time thus:

S4	S3	Active Segment Register
0	0	Extra segment
0	1	Stack segment
1	0	Code segment (or no segment)
1	1	Data segment

The S5 pin indicates the status of the interrupt enable flag.

The S6 pin becomes a logic LOW during the first part of the execution of I/O instructions.

Maximum and Minimum Modes

The 8086 has been designed to work in one of two modes. Simple single-processor applications use the MINIMUM mode. However, if multiple processors are to be used, then the 8086 should be in the MAXIMUM mode. The mode of operation is determined by the logic level presented at the MN/$\overline{\text{MX}}$ pin.

8086 MINIMUM MODE

A diagram of the basic MINIMUM mode system is shown in Figure 7.

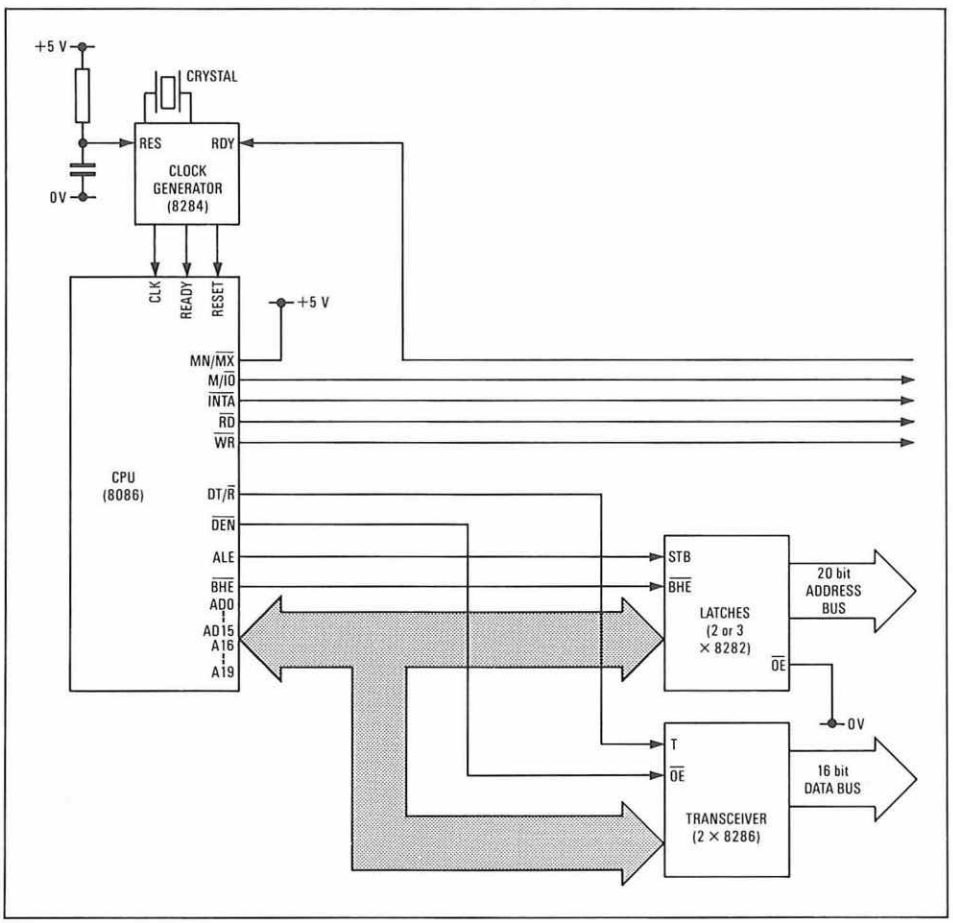


Figure 7
Minimum mode system

The data bus transceiver shown in Figure 7 may be omitted where bus loading is small.

In the MINIMUM mode, the control signals generated by the 8086 are almost the same as those generated by the 8085:

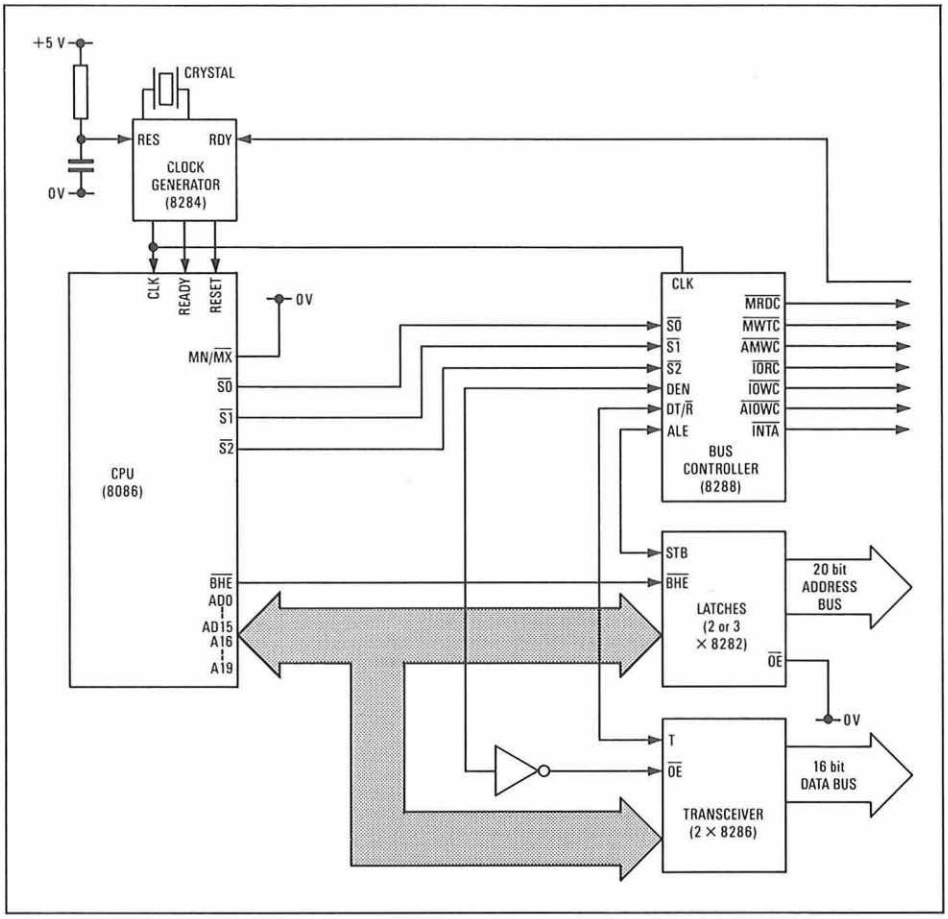
ALE M/$\overline{\text{IO}}$ $\overline{\text{WR}}$ DT/$\overline{\text{R}}$ $\overline{\text{DEN}}$ $\overline{\text{INTA}}$ HOLD HLDA

However, the system has a 16 bit data bus and a full 20 bit address bus. The pin functions described so far apply to the MINIMUM mode.

8086 MAXIMUM MODE

A diagram of the basic maximum mode system is shown in Figure 8.

Figure 8
Maximum mode system



In the MAXIMUM mode, the 8086 can operate with a co-processor (for example, the 8087 maths co-processor). This allows the sharing of system resources. This option is supported by a bus lock mechanism. The LOCK pin becomes a logic LOW when a LOCK prefix instruction is executed and remains LOW during the execution of the instruction immediately following. This can be used to prevent other processors from gaining bus control from the 8086 during the execution of a multibyte instruction.

The sophisticated facilities required by multiple processor operation necessitate the generation of further control signals. An 8288 bus controller is used to generate both the extra signals required and some of the control signals produced by the MINIMUM mode 8086. The S0, S1 and S2 outputs from the 8086 are decoded by the 8288 to give the following extra control signals:

$\overline{\text{MRDC}}$ Memory read control

$\overline{\text{MWTC}}$ Memory write control

$\overline{\text{AMWC}}$ Advanced memory write control

$\overline{\text{IORC}}$ I/O read control

$\overline{\text{IOWC}}$ I/O write control

$\overline{\text{AIOWC}}$ Advanced I/O write control

These signals are essentially concerned with handshaking for memory or I/O operations. The generation of the following minimum mode 8086 signals is also assumed by the 8288 bus controller:

DEN DT/$\overline{\text{R}}$ ALE $\overline{\text{INTA}}$

The status of the instruction queue is indicated by QS1 and QS0 thus:

QS1	QS0	Queue Status
0	0	No operation
0	1	Queue being emptied
1	0	Execution of first instruction byte
1	1	Removal from queue of subsequent instruction byte

BACK NUMBERS

The price of back numbers of the *Journal* including the *Supplement* and postage and packaging is £2 (UK); £2.50 (overseas). The price to staff of British Telecom and the British Post Office is 75p per copy. (The *Supplement* is not sold separately.) Back issues can be ordered by using the form printed below.

Details of the question/answer material and educational papers published in back issues of the *Supplement* are shown below. For each subject, a list of back issues in which that subject appeared is given. Please note that each issue of the *Supplement* contains several subjects. Take care not to order a particular issue twice.

BTEC UNITS

(Question and answer material based on BTEC syllabi)

Digital Techniques II	Apr. 1983, Apr. 1984, Apr. 1985, July 1986
Digital Techniques A III	Oct. 1983, July 1984, Oct. 1985, Oct. 1986
Electrical and Electronic Principles II	July 1983, Apr. 1984, Oct. 1985, Jan. 1987
Electrical and Electronic Principles III	Oct. 1983, Oct. 1984, Jan. 1986
Electrical Principles II	Jan. 1980, Jan. 1981, July 1982
Electronics II	Oct. 1980
Electronics III	July 1983, Apr. 1984, July 1985
Line and Customer Apparatus I	Jan. 1979, July 1981, Apr. 1982, Apr. 1983, Apr. 1984, Oct. 1986
Lines II	Apr. 1981, Oct. 1982, Apr. 1984, Apr. 1985, Apr. 1986, Apr. 1987
Lines III	July 1983, Apr. 1986
Mathematics I	Jan. 1979, Apr. 1980
Mathematics II	Apr. 1980, Apr. 1982, July 1983, Apr. 1984, July 1985, Apr. 1986
Micro-Electronic Systems I	July 1983, Apr. 1984, Apr. 1985
Micro-Electronic Systems II	Oct. 1983, July 1984, July 1985, July 1987
Physical Science I	Jan, 1979, Jan. 1980, July 1981
Radio II	Jan. 1983, Jan. 1984, Apr. 1985
Radio III	July 1986
Telecommunications Systems I	July 1981, July 1982, July 1983, Jan. 1984, Jan. 1985, July 1986
Telephone Switching Systems II	Jan. 1980, Oct. 1982, Oct. 1983, July 1984, Oct. 1985, Oct. 1986
Telephone Switching Systems III	Jan. 1983, Oct. 1984, Apr. 1986
Transmission Systems II	Jan. 1982, Jan. 1983, Jan. 1984, Jan. 1985, Jan. 1986, Jan. 1987
Transmission Systems III	Apr. 1983, Oct. 1985, Oct. 1986

CITY AND GUILDS OF LONDON INSTITUTE

(Answers to examination papers set by CGLI. Year of paper shown in brackets)

Circuit Theory T4	Apr. 1986 (1985), Apr. 1987 (1986)
Electrical Principles T3	Oct. 1986 (1985)
Electronics T3	Oct. 1986 (1985)
Electronics T4	Jan. 1987 (1985)
Microelectronic Systems T3 Option	Oct. 1986 (1985)
Microelectronic Systems T4 Option	July 1987 (1986)
Switching T4 Option	Jan. 1986 (1985), Jan. 1987 (1986)
Switching T5 Option	Apr. 1986 (1985), Apr. 1987 (1986)
Transmission T4 Option	July 1986 (1985), July 1987 (1986)
Transmission T5 Option	Apr. 1987 (1986)

EDUCATIONAL PAPERS PUBLISHED IN THE SUPPLEMENT

Field-Effect Transistors	Oct. 1982
Microcomputer Systems (Part 1)	Oct. 1984
Microcomputer Systems (Part 2)	Jan. 1985
Digital Multiplexing	Apr. 1986
The Purposes of Teletraffic Engineering and its Application	Oct. 1987
A Guideline for Writing a System Requirements Specification for Computer Systems	Jan. 1988
An Introduction to 16 bit Microprocessors Part 1—8086/8088 Microprocessors	Apr. 1988

Note: The January and April 1982, and the January and April 1985 back issues are no longer available, but photocopies of the *Supplements* can be supplied for the same price.

To: British Telecommunications Engineering (Sales),
Post Room, 2-12 Gresham St.,
London EC2V 7AG.

Please send the following back issues of *British Telecommunications Engineering* .....

.....

Name ..... Address .....

I enclose a cheque/postal order† for £ ....

.....

†Cheques and postal orders, payable to **'BTE Journal'**, should be crossed '& Co.'. Cash should not be sent through the post.

Published by *British Telecommunications Engineering Journal*, 2–12 Gresham Street, London EC2V 7AG
and printed in Great Britain by Unwin Brothers Limited, Old Woking, Surrey